OpenMP extensions for FPGA Accelerators

Daniel Cabrera[1,2], Xavier Martorell[1,2], Georgi Gaydadjiev[3], Eduard Ayguade[1,2], Daniel Jiménez-González[1,2]
[1]Barcelona Supercomputing Center
c/Jordi Girona 31,
Torre Girona,
E-08034 Barcelona, Spain

[2]Universitat Politecnica de Catalunya
c/Jordi Girona 1-3,
Campus Nord-UPC, Modul C6,
E-08034 Barcelona, Spain
{dcabrera, xavim, eduard, djimenez}@ac.upc.edu

[3]Delft University of Technology
Mekelweg 4,
2628 CD Delft,
The Netherlands
g.n.gaydadjiev@its.tudelft.nl

***Abstract*—Reconfigurable computing is one of the paths to explore towards low-power supercomputing. However, programming these reconfigurable devices is not an easy task and still requires significant research and development efforts to make it really productive. In addition, the use of these devices as accelerators in multicore, SMPs and ccNUMA architectures adds an additional level of programming complexity in order to specify the offloading of tasks to reconfigurable devices and the interoperability with current shared-memory programming paradigms such as OpenMP. This paper presents extensions to OpenMP 3.0 that try to address this second challenge and an implementation in a prototype runtime system. With these extensions the programmer can easily express the offloading of an already existing reconfigurable binary code (bitstream) hiding all the complexities related with device configuration, bitstream loading, data arrangement and movement to the device memory. Our current prototype implementation targets the SGI Altix systems with RASC blades (based on the Virtex 4 FPGA). We analyze the overheads introduced in this implementation and propose a hybrid host/device operational mode to hide some of these overheads, significantly improving the performance of the applications. A complete evaluation of the system is done with a matrix multiplication kernel, including an estimation considering different FPGA frequencies.**

I. Introduction

The gigahertz race to which we were used to in the last decade has stopped due to power dissipation problems. The extra transistors that are available for new designs are not used to increase the complexity of superscalar architectures, out of order or multithreaded. The technological increase in transistor count is used to include more that one core in the same chip (homogeneous multicore) and/or to incorporate accelerators (heterogeneous multicore) well suited for certain application domains, such as for example GPU units in [1] or vector units in the Cell/B.E.[2]. For these accelerators, the exploitation of the potential parallelism available is not an easy task, and relies on the use of specific SDKs.

The use of specialized devices designed to compute some specific function (ASIC circuits) is another alternative to benefit a specific kind of applications. For example an ASIC to compute *fast Fourier transforms* can clearly eliminate the computation bottlenecks found in some bioinformatics applications. Field Programmable Gate Arrays (FPGA) are accelerators whose specific functionality can be retargetted to different domains at runtime. However, efficiently programming these specific functionalities requires the use of low-level hardware description languages (HDL), such as Verilog or VHDL, to which general-purpose programmers are not used to.

The productive parallelization of applications for heterogenous multicore architectures that include one or more of such accelerators requires programming models able to express the proper of oading of tasks and the data that is needed to perform the computation. This is the purpose of this paper, and in particular, to show a proposal that extends OpenMP 3.0 tasking [3] to target heterogeneous architectures with FPGA-based accelerators. OpenMP 3.0 task pragmas completely fits with the idea of using one or more FPGAs as accelerators. In this paper we are assuming that the bitstreams that corresponds to the computations to be of oaded in tasks are either existing IP cores or are generated using other compilation tools. This may impose some restrictions in the behavior of the tasks to be of oaded, such as for example on the use of synchronization constructs.

In order to motivate our extensions to OpenMP 3.0 and their implementation in the runtime system, Figure 1 shows part of the code that is necessary to of oad the execution of a matrix multiplication bitstream *matmul_fpga* to one of the FPGAs available on the SGI RASC architecture [4], using the SGI RASClib library [5]. In addition to this, the programmer needs to change the memory association of data in the host when transfers to/from the FPGA device

978-1-4244-4501-1/09/$25.00 ©2009 IEEE

(pack/unpack), which may also require the use of blocking in order to fit the requirements of the accelerator bitstream and memory. Our proposed extensions and runtime implementation try to hide all these complexities, making the parallelization and of oading of tasks in accelerators more productive.

The rest of the paper is organized as follows: Section II presents the extensions proposed for OpenMP. Section III shows implementation details of the runtime system. Sections IV details the experimental setup. Sections V and VI show experimental evaluation. Section VII presents related work, and we conclude with Section VIII.

II. Proposed OpenMP extensions for FPGA

Tasks are the most important new feature of OpenMP 3.0. A programmer can define deferrable units of work, called tasks, and later ensure that all the tasks defined up to some point have finished.

```
#pragma omp task [clause-list]
   structured-block
```

Clauses can be used to specify data scoping (`shared`, `private` or `firstprivate`) and conditional execution as a task (`if`), mainly. The runtime system launches the execution of the code in `structured-block` in the scope of the parent task, following the data scoping attributes indicated. The current execution model assumes that a thread in the current team of threads will execute the task. The proposal in [6] extended the `task` construct with some additional clauses to derive dependencies among tasks at runtime

- `input(data-reference-list)`
- `output(data-reference-list)`
- `inout(data-reference-list)`

The information provided in these clauses will be used by the runtime system to analyze the dependencies among tasks and guarantee the proper order execution of them as proposed in [6]. Although in some cases the compiler can analyze the code and determine the input and output data sets, we provided these additional clauses to modify or augment the compiler analysis.

In an heterogeneous multicore architecture, we need some additional information in order to appropriately assign the execution of the task to any of the available devices, a GPU, an vector unit, FPGA device, ... Our proposals leverages on previous proposals that allow the specification of dependencies between tasks [6], loop blocking transformations specified at the pragma level [7], and the use of accelerators [8], all of them in the scope of OpenMP 3.0. In the following subsections we comment each one of the new pragmas that we use and/or extend in order to consider FPGA-based accelerator architectures.

Figure 2 shows the full version using our new pragmas in OpenMP 3.0 (pragmas details in Section II). Task of oading to `fpga` device is specified in the header function *matmul_fpga* (line 8 in Figure 2), and the appropiate blocking and packing/unpacking of data is expressed through the `block` pragma (line 17) and the specification of the direction of the arguments (line $7, 19, 20$). The runtime system will take care of efficiently implementing them and hiding their possible overheads.

```
1  void matrix_multiplication (float A[BS][BS],
2             float B[BS][BS],
3             float C[BS][BS])
4  {
5
6   /* Configure device */
7   res = rasclib_resource_configure("matmul_fpga",
8           num_devices, NULL);
9   algorithm_id = rasclib_algorithm_open("matmul_fpga",
10          RASCLIB_BUFFERED_IO);
11
12 /* Send inputs */
13  res = rasclib_algorithm_send(algorithm_id, "A",
14                               A, sizeof(A));
15
16  res = rasclib_algorithm_send(algorithm_id, "B",
17                               B, sizeof(B));
18 /* Execute */
19  rasclib_algorithm_go(algorithm_id);
20
21  /* Receive outputs */
22  res = rasclib_algorithm_receive(algorithm_id, "C",
23                   C, sizeof(C));
24
25  /* Commit commands and wait */
26  rasclib_algorithm_commit(algorithm_id, NULL);
27  rasclib_algorithm_wait(algorithm_id);
28  rasclib_resource_return("matmul_fpga", num_devices);
29
30 }
```

Fig. 1. Basic example programmed using RASClib. Error check has been omitted

```
1  float A[DIM_SIZE][DIM_SIZE];
2  float B[DIM_SIZE][DIM_SIZE];
3  float C[DIM_SIZE][DIM_SIZE];
4
5  #pragma omp target device(fpga) \
6      implements(block_matmul) \
7      copy_in(A,B) copy_inout(C)
8  extern void matmul_fpga(float A[BS][BS],
9              float B[BS][BS], float C[BS][BS]);
10
11 ...
12
13 int i, j, k;
14 for (i = 0; i < DIM_SIZE; i++) {
15    for (j = 0; j < DIM_SIZE; j++) {
16       for (k = 0; k < DIM_SIZE; k++) {
17         #pragma omp block nest(3) factor(BS,BS,BS) \
18         #pragma omp task label(block_matmul) \
19                          input(A[i][k],B[k][j]) \
20                          inout(C[i][j])
21         C[i][j] += A[i][k] * B[k][j];
22       }
23    }
24 }
25 ...
```

Fig. 2. Matrix multiplication using the proposed OpenMP extensions

A. Target device pragma

The following pragma [8] may precede any existing pragma task or function declaration or header

```
#pragma omp target device(device-name-list)
                          [clause-list]
   {pragma-task|
    function-definition|
    function-header}
```

It is used to specify that the execution of the task could be of oaded to any of the devices specified in `device-name-list`. The names used in this list are vendor specific (i.e. `cell`, `cuda`, ...). In the case of using FPGA accelerators, the `device-name` should be `fpga`. Then, when a task is ready for execution (i.e. it has no dependencies with other previously generated tasks) the runtime can choose among the different available targets to decide in which device to execute the task. If no resource is available (or not configured yet), the runtime could execute the default implementation on the host or stall until one of the resources becomes available.

Some additional clauses can be used with this pragma `device`:

- `copy_in(data-reference-list)`
  `copy_out(data-reference-list)`
  `copy_inout(data-reference-list)`
- `implements(function-name or label-name)`

The first three clauses, which are ignored for the shared-memory architectures, specify data movement for the shared variables used inside the task. `Copy_in` will move variables in `data-reference-list` from host to device memory. `Copy_out` will move variable in `data-reference-list` back from device to host memory. `Copy_inout` will do both. Once the task is ready for execution, the runtime system will move variables in the `copy_in` or `copy_inout` lists. Once the task finishes execution, the runtime will move variables in the `copy_out` or `copy_inout` lists, if necessary.

Clause `implements(function-name)` is used to specify an alternative implementation for a function that is invoked as a task. For instance, in the following code excerpt:

```
#pragma omp task input(A[BS][BS], B[BS][BS])
                 inout(C[BS][BS])
extern void matmul(float A[BSIZE][BSIZE],
            float B[BSIZE][BSIZE],
            float C[BSIZE][BSIZE]);

#pragma omp target device(fpga) \
            implements(matmul) \
            copy_in(A, B) copy_inout(C)
extern void matmul_fpga(float A[BS][BS],
            float B[BS][BS],
            float C[BS][BS]);
```

The programmer specifies that `matmul_fpga` is an alternative implementation of `matmul` when of oading the execution of that function to an FPGA device. In addition the programmer is specifying a change in memory association of the blocks used in the host implementation (`matmul`) and in the of oaded implementation (`matmul_fpga`). Notice that the accelerator version uses blocks of BSxBS contiguous elements, while in the host the block of BSxBS elements is part of a larger matrix of BSIZExBSIZE elements.

B. `label-name` clause

In order to allow the specification of a alternative implementations for structured code blocks, we extend the `task` pragma with an additional `label-name` clause. The `input`, `output` and `inout` clauses in `task` together with the `copy_in`, `copy_out` and `copy_inout` clauses in `target` are used to match variables used in the structured code block with arguments in the function used to implement it.

For the example shown in Figure 2, the programmer specifies that *matmul_fpga* bitstream will be used to of oad the execution of the structured code block *block_matmul* to the FPGA device.

In the case of function calls, it is not necessary to specify the `label` clause since the function name is used as label to identify alternative implementations.

C. Block pragma

When of oading the execution of tasks to accelerators, it is necessary to fit the problem to the limitations (for example memory) of those accelerators. In the case of FPGA accelerators, the specific hardware implementation of the task code in the FPGA can introduce additional constraints. In this paper we propose the use of pragmas to drive program transformations that are necessary to fit the task into the accelerator device. In particular the use of loop blocking, a well-known compiler technique used to optimize the exploitation of data locality.

The `block` pragma is introduced to block perfectly nested loops whose body is to be of oaded into the accelerator.

```
#pragma omp block nest(block-dimension)
            factor(block-size-list)
   task-code
```

The `nest(block-dimension)` clause specifies that `block-dimension` consecutive loops are affected by the blocking, being the inner loop the one that contains the `block` pragma. On the other hand, the `block-size-list` on the `factor` clause specifies the blocking size that should be used for each loop and induction variable. In addition to the loops, the pragma also transforms all the references in the `data-reference-lists` that are included in `input`, `output` and `inout` clauses.

To illustrate the effect of this pragma, Figure 3 shows how a source-to-source restructurer would transform the `block` pragma in Figure 2. The code transformation shown in Figure 3 should be manually applied by the programmer if the proposed `block` clause would not exist. This `block` pragma makes programming for such accelerators much more productive, avoiding the writing of the bounds of the block of elements that have to be moved to/from the accelerator.

Other more complex blocking strategies will have to be manually introduced by the programmer. However, the simple one proposed here is widely applied.

III. Runtime System Implementation

The runtime system should provide support to of oad the execution of bitstreams and the required data transfers for the SGI RASC technology following the OpenMP 3.0 pragma extensions described in Section II. In addition to this support, an implementation should also include the following main features:

- Bitstream cache and support for hybrid computation.
- Transparent change of memory association, providing data packing and unpacking when transferring data between host and FPGA device.
- Multithreaded FPGA library interface.

In the following subsections we detail a possible implementation of these features.

A. Bitstream cache and hybrid computation

As we will analyze in Section V, the time required to configure a bitstream in the FPGA can be significantly high.

```
1
2 #pragma omp target device(fpga) \
3             implements(block_matmul) \
4             copy_in(A, B) copy_inout(C)
5 extern void matmul_fpga(float A[BS][BS],
6                         float B[BS][BS],
7                         float C[BS][BS]);
8 ...
9
10 int i, j, k;
11 for (i = 0; i < DIM_SIZE; i+=BS)
12    for (j = 0; j < DIM_SIZE; j+=BS)
13       for (k = 0; k < DIM_SIZE; k+=BS) {
14 pragma omp task label(block_matmul) \
15          input(A[i:i+BS-1][k:k+BS-1], \
16            B[k:k+BS-1][j:j+BS-1]) \
17          inout(C[i:i+BS-1][j:j+BS-1])
18       {
19          for (i_b = i; i_b < i+BS; ib++)
20             for (j_b = j; j_b < j+BS; jb++)
21                for (k_b = k; k_b < k+BS; kb++)
22                   C[i_b][j_b] += A[i_b][k_b] * B[k_b][j_b];
23       }
24       }
25 ...
```

Fig. 3. Restructured version of the matrix multiplication using the proposed OpenMP extensions, after blocking

To avoid unnecessary configurations, the runtime system implements a fully associative cache structure to keep information about the bitstreams currently loaded in the FPGA devices. When a `task` pragma is found, the runtime checks if the bitstream that implements the function or structured code block associated with the task is already configured. A hit in the bitstream cache produces the effective of oading of the task execution. If the runtime detects a miss in the bitstream cache, it will apply a least frequently used (LFU) replacement policy, initializes the FPGA device associated to replace the bitstream and configure the FPGA device with the new bitstream. The number of entries of the bitstream cache is the number of FPGA devices we have. In case of having partial reconfigurations, the bitstream cache should take them into account.

In order to hide the FPGA configuration time that happens on a miss, the runtime applies what we call a hybrid computation policy. In this hybrid computation mode, when a miss occurs the runtime checks if the configuration for that task is already in progress; if it is not in progress, the runtime will launch the configuration of the bitstream. In both cases, the runtime will execute the task in the host processor overlapping with the configuration process. Once the bitstream is configured, the runtime will detect a hit in the bitstream cache and of oad the execution of future instances of that task to the FPGA device.

B. Memory association changes: pack/unpack

The data transfer bandwidth between the host and the FPGA device can be a bottleneck depending on the application characteristics, and the system you are running on. Our runtime system deals with this issue doing data packing and unpacking, as indicated by the different memory associations detected by the source-to-source compiler.

C. Multithreaded FPGA library interface

The FPGA library interface should be implemented using threads in order to avoid the application to be blocked in FPGA management operations. For instance, a thread can be configuring the FPGA device meanwhile another one is doing useful work in the host side (hybrid computation). There are one master and one worker threads. The master thread queues operations in a consumer-producer structure that the worker reads. The master thread will be blocked if the queued operation is synchronous, otherwise it will continue. The worker thread is in charge of performing the queued operations and will unblock the master thread when necessary.

IV. Experimental setup

Our experiments have been run on a SGI Altix 4700 system equipped with 128 Itanium processors. The system we used also includes a RASC RC100 FPGA blade (with two Xilinx Virtex 4 LX200). The prototype runtime system

targets the SGI RASC library 2.2 and Core Services [5]. The source-to-source code transformation process has been implemented as a new pass in the Mercurium infrastructure (version 1.2.1) developed at BSC [9]. We use *gcc* 4.1.2 as backend for compilation to binary code. For all the compilations we have used *-O3* as optimization level for the software part. Also, we have utilized all the Makefiles that SGI provides to compile the HDL codes and generate the configuration files. Those Makefiles use *Xilinx ISE 9.1* to generate the bitstream. All timing results are obtained using the *gettimeofday()* system call.

In our current prototype implementation, an abstract layer, that is easily adaptable to other FPGA based architectures, has been implemented to manage the FPGA devices. Currently, such a layer does not consider neither multiple FPGA accelerators, nor partial reconfigurations. Finally, our current runtime implementation is not dealing with task dependencies.

V. Implementation Analysis

We have tested our extensions and runtime with the RASC library examples (the data flow algorithm which is a pattern matching program and the simple algorithm which performs logic manipulations) to evaluate the communication and the programming model cost. Furthermore, we perform a detailed analysis of our implementation using a matrix multiplication kernel.

A. Communication Costs

Bitstream loading cost: We define the bitstream loading cost as the accumulated time of reading the bitstream file and the configuration of the reconfigurable device. For the $4MB$ size of the matrix multiplication bitstream this time is 1 second. In fact, this is a cost that you can save if you keep information of which bitstream is already configured in the FPGA device. Our runtime system keeps a bitstream cache with that information, reducing that time to $4ms$ (since there is some work to be done in any case).

In any case, the first time that the FPGA device has to be configured we will have to pay that large amount of time. Therefore, our runtime system performs an hybrid computation of the task that has to be of load to an FPGA device, in the sense, that if there is a bitstream cache miss, the runtime system will spawn a thread to configure the bitstream in the FPGA device meanwhile executing the same task in an available device target.

Host-SRAM Communication Bandwidth: In order to evaluate the DMA transfer bandwidth from host to the SRAM of the FPGA board in the SGI Altix System, we have done a modification of one of the examples of the RASClib library, the simple algorithm which performs logic manipulation. These measures have been taken using the RASC driver in BUFFERED_IO mode. The modified bitstream reads the last element of a large chunk of data that has to be written in the memory of the FPGA and, when that value is valid, the algorithm finalizes, signaling the host part of the program. Using this bitstream we have a very good approximation of the real DMA transfer bandwidth between host and the SRAM of the FPGA: $0.8GB/s$.

We have also evaluated the bandwidth of the non-DMA transfers measuring their execution time. The RASClib allows us to send a command to the FPGA using a register. The function that performs this operation finishes when the data has been sent. The non-DMA data transfer bandwidth is $630KB/s$.

Runtime System Operation	Time (ms)	Executions
Initialization FPGA	13.562	1
Spawn to FPGA	4.651	n
Write Input Data to FPGA	0.186	n
Reset of FPGA	0.113	n
Write InOut Data to FPGA	0.112	n
Wait Execution FPGA	11.602	n
Read InOut Data from FPGA	0.111	n
Unload Bitstream	0.547	n
Uninit FPGA	0.090	1

TABLE I
RUNTIME SYSTEM OVERHEAD FOR PATTERN MATCHING ALGORITHM EXAMPLE

B. Programming Model Costs

In Table I we can see the operations used by our runtime system. Last column specifies the number of times that this operation has to be performed in the application: 1 for configuration/deconfiguration operations and n for application dependent operations, where n is equal or larger than 1. Results are for the pattern matching example that comes with the RASCLib library. As we can see, the most time consuming operations are: initialization of the FPGA device, spawn of the bitstream to the FPGA, and waiting for the FPGA to finish. The initialization time is not significant since it is only done once, and the waiting for the FPGA depends on the application. Finally, the spawn to the FPGA time shown in the Table is the bitstream loading cost commented above, when there is a hit in the bitstream cache implemented in our runtime system.

In the case of using the blocking technique, we may have to pay a cost of packing/unpacking blocks of consecutive memory to send/receive them to/from the FPGA device. Figure 4 shows the packing/unpacking process of a data block of matrix C. That cost is host architecture dependend. In the case of our matrix multiplication, using blocking, we have analyzed which is the overall application cost of packing/unpacking when doing a matrix multiplication of 256x256 elements in the SGI Altix 4700 machine. Table II shows the overall execution times of packing/unpacking of the data varying the block size from 32x32 upto 128x128, for a 256x256 matrix multiplication. The larger the block,

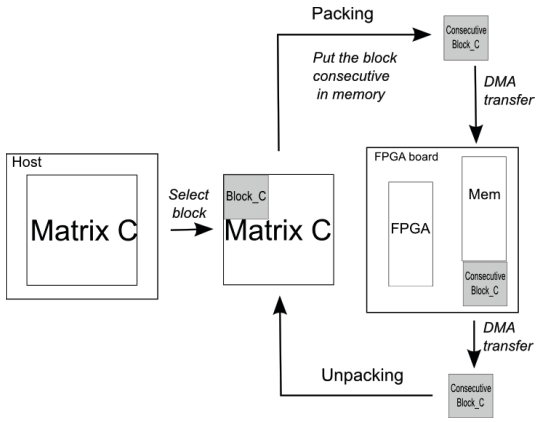

Fig. 4. Packing/Unpacking process for a data block of matrix C.

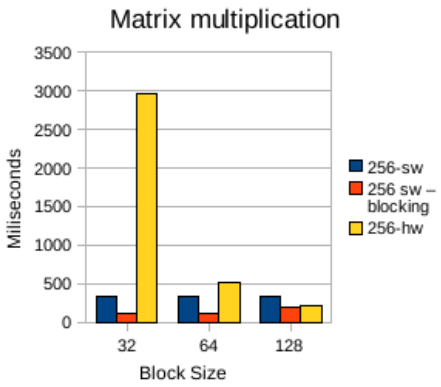

Fig. 5. Blocking comparison

Block size	Pack/Unpack (ms)
32x32	7.9
64x64	2.8
128x128	2.1

TABLE II
OVERALL EXECUTION TIME OF PACKING/UNPACKING BLOCKS IN THE MATRIX MULTIPLICATION OF 256X256 ELEMENTS

Block size	rasclib_algorithm_open (ms)
32x32	2160.2
64x64	271.6
128x128	35

TABLE III
TIME TO PERFORM THE RASCLIB_ALGORITHM_OPEN 256X256 ELEMENTS

the better the performance we achieve since we save block data management overheads, which is very significant for 32x32 blocks. Other memory transfer strategies to process all the blocks of whole matrix may help to improve the performance of the matrix multiplication.

VI. CASE OF STUDY: MXM

The objective of this section is not a performance evaluation of the FPGA in the SGI Altix 4700, neither a comparison between two different architectures, but to show a real example using our proposed extensions.

We have used a matrix multiplication core for up to 128x128 double precision elements, which works at 50 MHz in the FPGA. This is a version of the MxM in [10] adapted to the RASC interface. We use this core in order to perform a hardware blocked matrix multiplication using our blocking directives as shown in Section II.

The blocking directives let us use the matrix multiplication core as a block matrix multiplication, being able to multiply matrices larger than 128x128 elements (also smaller). Figure 2 shows the code we use in the evaluation.

Figure 5 shows execution time for three different versions of a 256x256 matrix multiplication: software version (256 sw), software version using blocking (256 sw-blocking), and blocking FPGA version (256 hw). In the case of the blocking version, block sizes of 32x32, 64x64 and 128x128 have been tested. The blocking software versions show better results than hardware version. Some of the reasons are: the extra overhead to execute in the FPGA, and the frequency of the matrix multiplication bitstream (50MHz). However, the execution time for the case of 32x32 blocks is very large and we analyzed it in detail. We figured out that the *rasclib_algorithm_open* function, used in the spawn of the thread to the FPGA to start executing the code, increases its execution time in each invocation. We cannot give any reason why this function is wasting that time since we do not have access to the SGI RASClib code. Table III shows the overhead produced by the *rasclib_algorithm_open* function. However, although any mechanism to reduce the number of times *rasclib_algorithm_open* is called will improve the overall performance of the matrix multiplication, the packing/unpacking overhead cost (shown in Table II), and other overhead costs per block, will still exist. Therefore, the best performance is achieved for the largest block size tested.

Finally, we wanted to evaluate the performance of the blocked hardware version if we have had a matrix multiplication with higher frequency than 50MHz. Therere, we have estimated the execution times of the matrix multiplication using 100MHz and 200MHz frequencies, properly scaling the execution time of the FPGA execution part. Figure 6 shows the execution time of the hardware version when the frequency is 50MHz (the real frequency of our matrix multiplication core), 100MHz and 200MHz. Results show that hardware version with larger frequencies will overcome the software version (*256 sw-blocking* in the Figure).

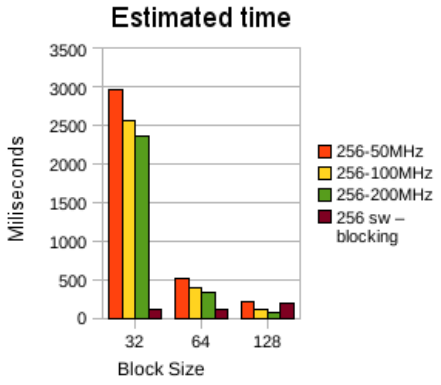

Fig. 6. Matrix multiplication estimated execution times.

VII. Related work

Hardware description languages (HDL), like *Verilog* and and *VHDL*, are employed to develop programs for FPGA Those languages are difficult to use for the software developers because programming should be spatial programming and at a very low level; being hard to debug.

In order to solve these problems of programming, new models have appeared. Most of these models are C extensions. There are many of them: *Impulse C* [11], *ROCCC* [12], *Streams C* [13], *Mitrion C* [14], *Molen programming paradigm* [15] etc. In these models, the programmer uses C language on those sections of code that should be of oaded the FPGA as a task. Usually, there are two strategies to accelerated/of oad a section of code. In the first strategy, the section of code to be of oaded to the FPGA is translated from C to VHDL. This strategy is followed by Streams C, Impulse C and ROCCC. In all of these models we have a software/hardware solution where the developers only have to use a C subset and the API to do the communications. More in detail, Streams C and Impulse C are focused on the communications between the core running on the FPGA and the application. In both models, the authors provide us an API for communications, allowing the developers to program communications like C streams. ROCCC is oriented to HPC, focusing on loop optimizations and how to exploit the parallelism of them using reconfigurable computing.

The second strategy is to map a soft processor into the FPGA, and translate the source code to be executed to the code that this soft processor understands. This strategy is followed by Mitrion C, where the compiler generates code for a Mitrion Virtual Processor (a customizable processor created by Mitrionics). In both strategies the compiler manages communications to improve transparency in programming. In our work, we do not generate code from C to any HDL or any assembler for a soft core. We use the OpenMP standard to use existings bitstreams in the applications.

On the other hand, there are other programming models that are close to our proposal (HMPP [16] and PGI [17]) in the sense of of oading tasks to accelerators. However, their current release just supports GPU accelerating coprocessors.

The HMPP [16] approach is to declare, by means of directives, functions (named codelets) suitable for hardware acceleration and callsites to them. It also includes directives to help the generator to produce efficient code (e.g. unroll-and-jam to increase register exploitation and decrease the number of loads and stores per operation). At execution, the HMPP runtime takes care of discovering the attached accelerators and their availability. When a codelet is indicated to be run on an accelerator, if the device is available and if the shared library corresponding codelet is present, HMPP loads it just as a software plug-in. Otherwise the native version is run on the host CPU or in a worker thread. The HMPP approach is quite similar to our proposal, since it also annotates functions to be loaded in the accelerators that are specified in the target clause. However, we think that our approach is better integrated in the OpenMP specification and supports hybrid computation of tasks so that the runtime can overlap computation of the task on the host CPU meanwhile the accelerator is being configured. Our also proposal integrates loop blocking pragmas that naturally interact with the clauses used to express data movement.

The directives and programming model defined by PGI [17] allow programmers to specify the regions of a host program (mainly loops) to be targeted for loading to an accelerator device, without the need to explicitly initialize the accelerator and manage data or program transfers between the host and accelerator. PGI is based on compiler technology to optimize the loading of loops in accelerators and the data movement between the memories, not by the runtime system like in our proposal.

In [18], [19] the authors define which directives from OpenMP 2.0 can be synthetized. Furthermore, they implement a framework that generates *SystemC* code from OpenMP directives. This code is simulated but they do not generate real hardware from these pragmas because they cannot be synthetized. Our approach is orthogonal to the ideas on that work as we are mostly focused on the new task pragma of OpenMP 3.0.

VIII. Conclusions

In this paper, we have presented a runtime system implementation of the OpenMP 3.0 extensions to of oad tasks to different target devices. In particular, we have implemented those extensions inside the Mercurium compiler framework in a SGI Altix architecture to use FPGA devices. Those OpenMP extensions helps programmability and reduce developing cost.

We have seen some bottlenecks on the SGI Altix architecture and environment, like the DMA transfer bandwidth

and the *rasclib_algorithm_open* function, that penalize the overall performance of the application. We will improve our bandwidth using using DIRECT_IO mode in the RASC driver in a future release. Another time consuming function of RASClib library is *rasclib_resource_con gure()*. This function has to read the bitstream from the hard disk, load it in the main memory, send it through the NUMAlink connection and configure the Virtex 4 FPGA. In particular, we have observed that most of the time is spent reading from the hard disk and configuring the FPGA. Our runtime system hides this time using hybrid computation and a bitstream cache. This hybrid computation allows to execute the host (software) version of the code to be of oaded meanwhile the bitstream is configured. That has been done using the bitstream cache in the runtime system to avoid unnecessary configurations. That process is transparent to the programmer and significantly improves the performance of the application. Our experiments show that hybrid computation of the task can significantly improve the use of target devices whose initialization cost is large.

Finally, we have done an evaluation of the bottlenecks of SGI Altix architecture and programming model implementation using several kernels, showing some results for a blocked matrix multiplication on the SGI Altix System.

A. Current and Future work

We are currently working on supporting several FPGAs in the SGI Altix System. Therefore, future implementations will consider multiple FPGA accelerators, or multiple accelerators in an FPGA and then, the runtime system will use those parameters to properly schedule the task executions.

As future work, we plan to get involved on the definition of the standard CPU–FPGA interface (CoreLib [20]), to connect bitstreams to the code generated by our compiler. Our idea is to use a toolchain from C to bitstream for those parts of code that will be mapped to the FPGA device.

Other future lines are the optimization of the data movements so that the runtime system can advance them (prefetching), new packing/unpacking techniques, runtime partial reconfiguration, OpenMP extensions to synchronize CPU and FPGA threads, etc.

ACKNOWLEDGMENTS

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (no. TIN2007-60625, CSD2007-00050 and TIN2006-27664-E), the European Commission in the context of the SARC project (no. 27648) and the HiPEAC Network of Excellence (no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement. We also wants to thank Mihai Lefter, from Delft University, for providing us the matrix multiplication bitstream used in some of our evaluations. Also, special thanks to Roger Ferrer for providing us Mercurium compiler and for solving all the questions about the compiler.

REFERENCES

[1] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, August 2008.

[2] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, and et al., "The Design and Implementation of a First-Generation Cell Processor," in *IEEE International Solid-State Circuits Conference (ISSCC 2005)*, 2005.

[3] "The OpenMP API specification," www.openmp.org/wp/openmp-specifications/, urls visited on April 3rd, 2009.

[4] "SGI Altix 4700," http://www.sgi.com/products/servers/altix/4000/.

[5] "SGI RASC Guide," http://techpubs.sgi.com/library/manuals/4000/007-4718-007/pdf/007-4718-007.pdf.

[6] A. Duran, J. M. Pérez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the openmp tasking model to allow dependent tasks," in *4th International Workshop on OpenMP, IWOMP 2008, West Lafayette, IN, USA, May 12-14, 2008*, ser. Lecture Notes in Computer Science, vol. 5004, pp. 111–122.

[7] R. Ferrer, M. Gonzalez, F. Silla, X. Martorell, and E. Ayguade, "Evaluation of memory performance on the cell be with the sarc programming model," in *Proceedings of the 9th Workshop on Memory Performance: Dealing with Applications, systems, and architecture (MEDEA 08), Toronto, Canada, October 2008*.

[8] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Ortiz, "A proposal to extend the openmp tasking model for heterogeneous architectures," in *To appear in 5th International Workshop on OpenMP, IWOMP 2009*.

[9] R. Ferrer, M. Gonzalez, X. Martorell, and E. Ayguade, "Mercurium c/c++ source-to-source compiler."

[10] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit oating-point fpga matrix multiplication," in *FPGA 05*. New York, NY, USA: ACM, 2005, pp. 86–95.

[11] "Impulse C website," http://www.impulsec.com/.

[12] W. A. Najjar, "Compiling code accelerators for fpgas," in *CASES 07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2007, pp. 1–2.

[13] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, p. 49, 2000.

[14] "Mitrion-c," http://www.mitrionics.com.

[15] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. M. Panainte, "The molen programming paradigm," in *in Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, 2003, pp. 1–10.

[16] S. B. R. Dolbeau and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[17] T. P. Group, "PGI Fortran and C Accelerator Compilers and Programming Model Technology Preview."

[18] P. Dziurzanski and V. Beletskyy, "Defining synthesizable openmp directives and clauses," in *International Conference on Computational Science*, 2004, pp. 398–407.

[19] P. Dziurzanski, W. Bielecki, K. Trifunovic, and M. Kleszczonek, "A system for transforming an ansi c code with openmp directives into a systemc description," in *DDECS 06: Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and systems*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 151–152.

[20] M. J. Wirthlin, D. S. Poznanovic, P. Sundararajan, A. J. Coppola, D. Pellerin, W. A. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov, "OpenFPGA CoreLib core library interoperability effort." *Parallel Computing*, vol. 34, no. 4-5, pp. 231–244, 2008.